Chapter 1. A Modern Language

The greatest challenges and most exciting opportunities for software developers today lie in harnessing the power of networks. Applications created today, whatever their intended scope or audience, will almost certainly run on machines linked by a global network of computing resources. The increasing importance of networks is placing new demands on existing tools and fueling the demand for a rapidly growing list of completely new kinds of applications.

We want software that works—consistently, anywhere, on any platform—and that plays well with other applications. We want dynamic applications that take advantage of a connected world, capable of accessing disparate and distributed information sources. We want truly distributed software that can be extended and upgraded seamlessly. We want intelligent applications that can roam the Net for us, ferreting out information and serving as electronic emissaries. We have known for some time what kind of software we want, but it is really only in the past few years that we have begun to get it.

The problem, historically, has been that the tools for building these applications have fallen short. The requirements of speed and portability have been, for the most part, mutually exclusive, and security has been largely ignored or misunderstood. In the past, truly portable languages were bulky, interpreted, and slow. These languages were popular as much for their high-level functionality as for their portability. Fast languages usually provided speed by binding themselves to particular platforms, so they met the portability issue only halfway. There were even a few safe languages, but they were primarily offshoots of the portable languages and suffered from the same problems. Java is a modern language that addresses all three of these fronts: portability, speed, and security. This is why it has been a dominant language in the world of programming for more than a decade and a half.

Enter Java

The Java programming language, developed at Sun Microsystems under the guidance of Net luminaries James Gosling and Bill Joy, was designed to be a machine-independent

programming language that is both safe enough to traverse networks and powerful enough to replace native executable code. Java addresses the issues raised here and played a starring role in the growth of the Internet, leading to where we are today.

Initially, most of the enthusiasm for Java centered on its capabilities for building embedded applications for the Web called *applets*. But in the early days, applets and other client-side GUI applications written in Java were limited. Today, Java has Swing, one of the most sophisticated toolkits for building graphical user interfaces (GUIs) in any language. This development has allowed Java to become a popular platform for developing traditional client-side application software.

Of even more importance, however, Java has become the premier platform for web-based applications and web services. These applications use technologies including the Java Servlet API, Java web services, and many popular open source and commercial Java application servers and frameworks. Java's portability and speed make it the platform of choice for modern business applications. Java servers running on open source Linux platforms are at the heart of the business and financial world today.

This book will show you how to use Java to accomplish real-world programming tasks. In the coming chapters we'll cover everything from text processing to networking, building rich client-side GUI applications with Swing and lightweight web-based applications and services.

Java's Origins

The seeds of Java were planted in 1990 by Sun Microsystems patriarch and chief researcher Bill Joy. At the time, Sun was competing in a relatively small workstation market while Microsoft was beginning its domination of the more mainstream, Intel-based PC world. When Sun missed the boat on the PC revolution, Joy retreated to Aspen, Colorado, to work on advanced research. He was committed to the idea of accomplishing complex tasks with simple software and founded the aptly named Sun Aspen Smallworks.

Of the original members of the small team of programmers assembled in Aspen, James Gosling will be remembered as the father of Java. Gosling first made a name for himself in the early 80s as the author of Gosling Emacs, the first version of the popular Emacs editor that was written in C and ran under Unix. Gosling Emacs became popular but was soon eclipsed by a free version, GNU Emacs, written by Emacs's original designer. By

that time, Gosling had moved on to design Sun's NeWS, which briefly contended with the X Window System for control of the Unix GUI desktop in 1987. Although some people would argue that NeWS was superior to X, NeWS lost because Sun kept it proprietary and didn't publish source code while the primary developers of X formed the X Consortium and took the opposite approach.

Designing NeWS taught Gosling the power of integrating an expressive language with a network-aware windowing GUI. It also taught Sun that the Internet programming community will ultimately refuse to accept proprietary standards, no matter how good they may be. The seeds of Java's licensing scheme and open (if not quite "open source") code were sown by NeWS's failure. Gosling brought what he had learned to Bill Joy's nascent Aspen project. In 1992, work on the project led to the founding of the Sun subsidiary FirstPerson, Inc. Its mission was to lead Sun into the world of consumer electronics.

The FirstPerson team worked on developing software for information appliances, such as cellular phones and personal digital assistants (PDAs). The goal was to enable the transfer of information and real-time applications over cheap infrared and traditional packet-based networks. Memory and bandwidth limitations dictated small, efficient code. The nature of the applications also demanded they be safe and robust. Gosling and his teammates began programming in C++, but they soon found themselves confounded by a language that was too complex, unwieldy, and insecure for the task. They decided to start from scratch, and Gosling began working on something he dubbed "C++ minus minus."

With the foundering of the Apple Newton (Apple's earliest handheld computer), it became apparent that the PDA's ship had not yet come in, so Sun shifted FirstPerson's efforts to interactive TV (ITV). The programming language of choice for ITV set-top boxes was to be the near ancestor of Java, a language called Oak. Even with its elegance and ability to provide safe interactivity, Oak could not salvage the lost cause of ITV at that time. Customers didn't want it, and Sun soon abandoned the concept.

At that time, Joy and Gosling got together to decide on a new strategy for their innovative language. It was 1993, and the explosion of interest in the Web presented a new opportunity. Oak was small, safe, architecture-independent, and object-oriented. As it happens, these are also some of the requirements for a universal, Internet-savvy programming language. Sun quickly changed focus, and, with a little retooling, Oak became Java.

Growing Up

It would not be overstating it to say that Java caught on like wildfire. Even before its first official release when Java was still a nonproduct, nearly every major industry player had jumped on the Java bandwagon. Java licensees included Microsoft, Intel, IBM, and virtually all major hardware and software vendors. However, even with all this support Java took a lot of knocks and experienced some growing pains during its first few years.

A series of breach of contract and antitrust lawsuits between Sun and Microsoft over the distribution of Java and its use in Internet Explorer hampered its deployment on the world's most common desktop operating system—Windows. Microsoft's involvement with Java also become one focus of a larger federal lawsuit over serious anticompetitive practices at the company, with court testimony revealing concerted efforts by the software giant to undermine Java by introducing incompatibilities in its version of the language. Meanwhile, Microsoft introduced its own Java-derived language called C# (C-sharp) as part of its .NET initiative and dropped Java from inclusion in Windows. C# has gone on to become a very good language in its own right, enjoying more innovation in recent years than has Java.

But Java continues to spread on a wide variety of platforms. As we begin looking at the Java architecture, you'll see that much of what is exciting about Java comes from the self-contained, virtual machine environment in which Java applications run. Java was carefully designed so that this supporting architecture can be implemented either in software, for existing computer platforms, or in customized hardware. Hardware implementations of Java are used in some smart cards and other embedded systems. You can even buy "wearable" devices, such as rings and dog tags, that have Java interpreters embedded in them. Software implementations of Java are available for all modern computer platforms down to portable computing devices. Today, an offshoot of the Java platform is the basis for Google's Android operating system that powers billions of phones and other mobile devices.

In 2010, Oracle corporation bought Sun Microsystems and became the steward of the Java language. In a somewhat rocky start to its tenure, Oracle sued Google over its use of the Java language in Android and lost. In July of 2011, Oracle released Java SE 7, a significant Java release.

A Virtual Machine

Java is both a compiled and an interpreted language. Java source code is turned into simple binary instructions, much like ordinary microprocessor machine code. However, whereas C or C++ source is reduced to native instructions for a particular model of processor, Java source is compiled into a universal format—instructions for a *virtual machine*.

Compiled Java *bytecode* is executed by a Java runtime interpreter. The runtime system performs all the normal activities of a hardware processor, but it does so in a safe, virtual environment. It executes a stack-based instruction set and manages memory like an operating system. It creates and manipulates primitive data types and loads and invokes newly referenced blocks of code. Most importantly, it does all this in accordance with a strictly defined open specification that can be implemented by anyone who wants to produce a Java-compliant virtual machine. Together, the virtual machine and language definition provide a complete specification. There are no features of the base Java language left undefined or implementation-dependent. For example, Java specifies the sizes and mathematical properties of all its primitive data types rather than leaving it up to the platform implementation.

The Java interpreter is relatively lightweight and small; it can be implemented in whatever form is desirable for a particular platform. The interpreter may be run as a separate application or it can be embedded in another piece of software, such as a web browser. Put together, this means that Java code is implicitly portable. The same Java application bytecode can run on any platform that provides a Java runtime environment, as shown in Figure 1-1. You don’t have to produce alternative versions of your application for different platforms, and you don’t have to distribute source code to end users.

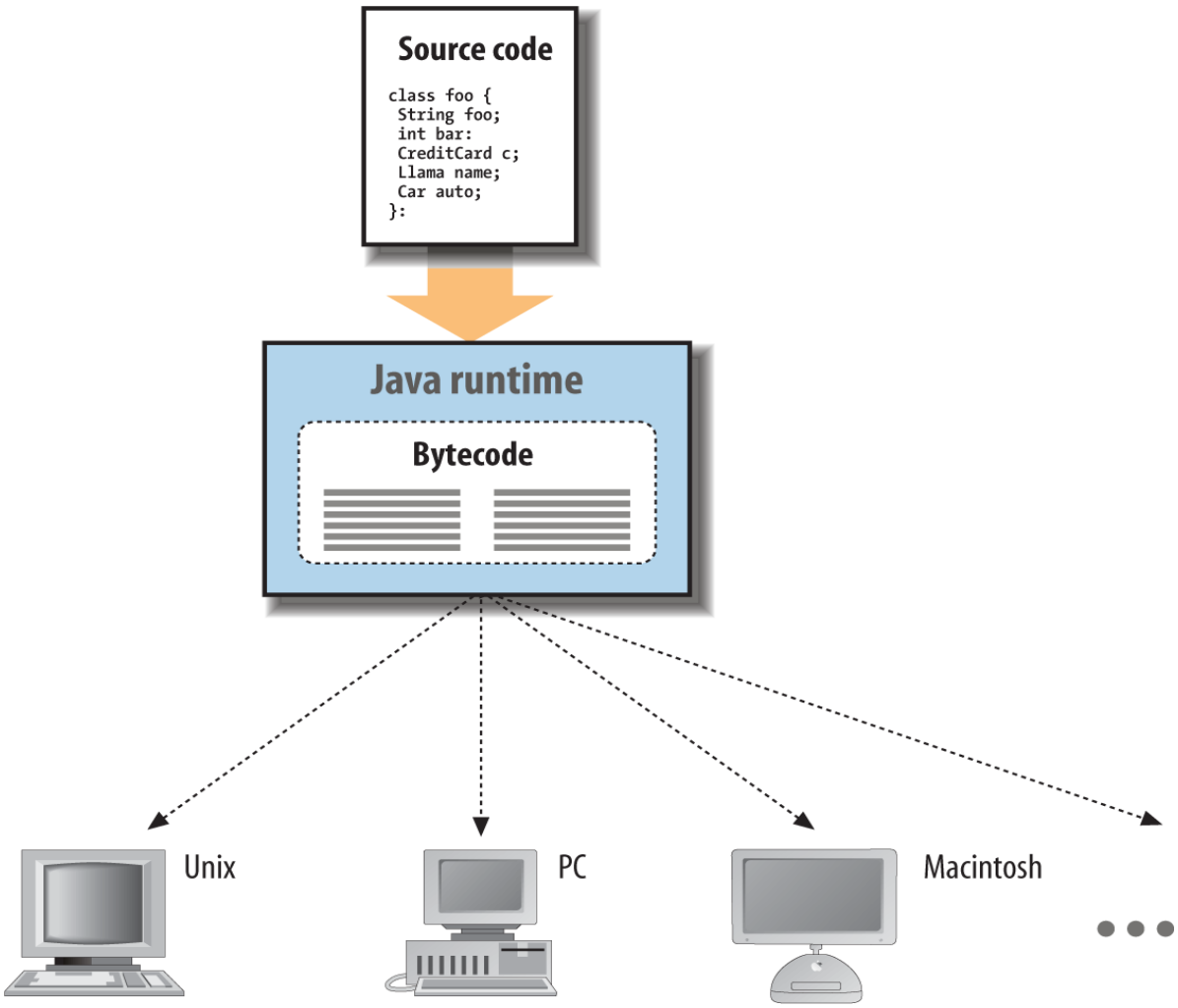

Figure 1-1. The Java runtime environment

The fundamental unit of Java code is the *class*. As in other object-oriented languages, classes are application components that hold executable code and data. Compiled Java classes are distributed in a universal binary format that contains Java bytecode and other class information. Classes can be maintained discretely and stored in files or archives locally or on a network server. Classes are located and loaded dynamically at runtime as they are needed by an application.

In addition to the platform-specific runtime system, Java has a number of fundamental classes that contain architecture-dependent methods. These *native methods* serve as the gateway between the Java virtual machine and the real world. They are implemented in a natively compiled language on the host platform and provide low-level access to resources such as the network, the windowing system, and the host filesystem. The vast majority of Java, however, is written in Java itself—bootstrapped from these basicprimitives—and is therefore portable. This includes fundamental Java tools such as

the Java compiler, networking, and GUI libraries, which are also written in Java and are therefore available on all Java platforms in exactly the same way without porting.

Historically, interpreters have been considered slow, but Java is not a traditional interpreted language. In addition to compiling source code down to portable bytecode, Java has also been carefully designed so that software implementations of the runtime system can further optimize their performance by compiling bytecode to native machine code on the fly. This is called just-in-time (JIT) or dynamic compilation. With JIT compilation, Java code can execute as fast as native code and maintain its transportability and security.

This is an often misunderstood point among those who want to compare language performance. There is only one intrinsic performance penalty that compiled Java code suffers at runtime for the sake of security and virtual machine design—array bounds checking. Everything else can be optimized to native code just as it can with a statically compiled language. Going beyond that, the Java language includes more structural information than many other languages, providing for more types of optimizations. Also remember that these optimizations can be made at runtime, taking into account the actual application behavior and characteristics. What can be done at compile time that can't be done better at runtime? Well, there is a tradeoff: time.

The problem with a traditional JIT compilation is that optimizing code takes time. So a JIT compiler can produce decent results, but may suffer a significant latency when the application starts up. This is generally not a problem for long-running server-side applications, but is a serious problem for client-side software and applications that run on smaller devices with limited capabilities. To address this, Java's compiler technology, called HotSpot, uses a trick called *adaptive compilation*. If you look at what programs actually spend their time doing, it turns out that they spend almost all their time executing a relatively small part of the code again and again. The chunk of code that is executed repeatedly may be only a small fraction of the total program, but its behavior determines the program's overall performance. Adaptive compilation also allows the Java runtime to take advantage of new kinds of optimizations that simply can't be done in a statically compiled language, hence the claim that Java code can run faster than C/C++ in some cases.

To take advantage of this fact, HotSpot starts out as a normal Java bytecode interpreter, but with a difference: it measures (profiles) the code as it is executing to see what parts are being executed repeatedly. Once it knows which parts of the code are crucial to

performance, HotSpot compiles those sections into optimal native machine code. Since it compiles only a small portion of the program into machine code, it can afford to take the time necessary to optimize those portions. The rest of the program may not need to be compiled at all—just interpreted—saving memory and time. In fact, the Java VM can run in one of two modes: client and server, which determine whether it emphasizes quick startup time and memory conservation or flat out performance.

A natural question to ask at this point is, Why throw away all this good profiling information each time an application shuts down? Well, Sun partially broached this topic with the release of Java 5.0 through the use of shared, read-only classes that are stored persistently in an optimized form. This significantly reduced both the startup time and overhead of running many Java applications on a given machine. The technology for doing this is complex, but the idea is simple: optimize the parts of the program that need to go fast and don't worry about the rest.

Java Compared with Other Languages

Java draws on many years of programming experience with other languages in its choice of features. It is worth taking a moment to compare Java at a high level with some other languages, both for the benefit of those of you with other programming experience and for the newcomers who need to put things in context. We do not expect you to have a knowledge of any particular programming language in this book and when we refer to other languages by way of comparison, we hope that the comments are self-explanatory.

At least three pillars are necessary to support a universal programming language today: portability, speed, and security. Figure 1-2 shows how Java compares to a a few of the languages that were popular when it was created.

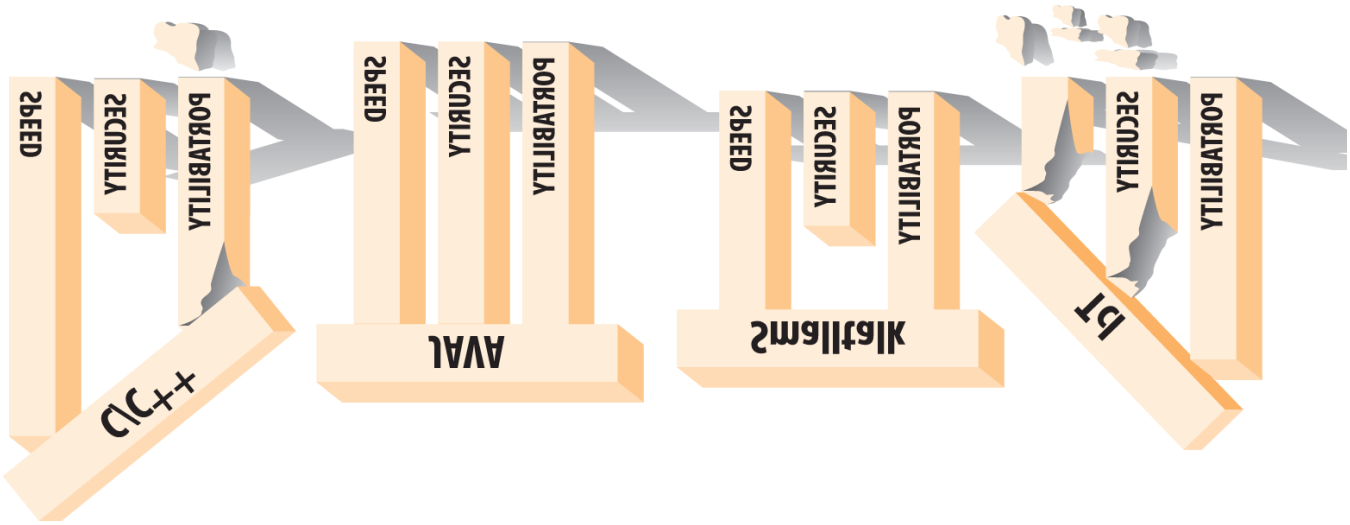

Figure 1-2. Programming languages compared

You may have heard that Java is a lot like C or C++, but that's really not true except at a superficial level. When you first look at Java code, you'll see that the basic syntax looks like C or C++. But that's where the similarities end. Java is by no means a direct descendant of C or a next-generation C++. If you compare language features, you'll see that Java actually has more in common with highly dynamic languages such as Smalltalk and Lisp. In fact, Java's implementation is about as far from native C as you can imagine.

If you are familiar with the current language landscape, you will notice that C#, a popular language, is missing from this comparison. C# is largely Microsoft's answer to Java, admittedly with a number of niceties layered on top. Given their common design goals and approach (e.g., use of a virtual machine, bytecode, sandbox, etc.), the platforms don't differ substantially in terms of their speed or security characteristics. C# is theoretically as portable as Java, but to date it is supported on far fewer platforms. Like Java, C# borrows heavily from C syntax but is really a closer relative of the dynamic languages. Most Java developers find it relatively easy to pick up C# and vice versa. The majority of time spent moving from one to the other is learning the standard library.

The surface-level similarities to these languages are worth noting, however. Java borrows heavily from C and C++ syntax, so you'll see terse language constructs, including an abundance of curly braces and semicolons. Java subscribes to the C philosophy that a good language should be compact; in other words, it should be sufficiently small and regular so a programmer can hold all the language's capabilities in his or her head at once. Just as C is extensible with libraries, packages of Java classes can be added to the core language components to extend its vocabulary.

C has been successful because it provides a reasonably feature-packed programming environment, with high performance and an acceptable degree of portability. Java also tries to balance functionality, speed, and portability, but it does so in a very different way. C trades functionality for portability; Java initially traded speed for portability. Java also addresses security issues that C does not (although in modern systems many of those concerns are now addressed in the operating system and hardware).

In the early days before JIT and adaptive compilation, Java was slower than statically compiled languages and there was a constant refrain from detractors that it would never catch up. But as we described in the previous section, Java's performance is now comparable to C or C++ for equivalent tasks and those criticisms have generally fallen quiet. ID Software's open source Quake2 video game engine has been ported to Java. If Java is fast enough for first-person combat video games, it's certainly fast enough for business applications.

Scripting languages such as Perl, Python, and Ruby are very popular. There's no reason a scripting language can't be suitable for safe, networked applications. But most scripting languages are not well suited for serious, large-scale programming. The attraction to scripting languages is that they are dynamic; they are powerful tools for rapid development. Some scripting languages such as Perl also provide powerful tools for text-processing tasks that more general-purpose languages find unwieldy. Scripting languages are also highly portable, albeit at the source code level.

Not to be confused with Java, JavaScript is an object-based scripting language originally developed by Netscape for the web browser. It serves as a web browser resident language for dynamic, interactive web-based applications. JavaScript takes its name from its integration with and similarities to Java, but the comparison really ends there. While there have been applications of JavaScript outside of the browser, it has not truly caught on as a general scripting language. For more information on JavaScript, check out*JavaScript: The Definitive Guide* by David Flanagan (O'Reilly).

The problem with scripting languages is that they are rather casual about program structure and data typing. Most scripting languages (with a hesitant exception for Python and later versions of Perl) are not object-oriented. They also have simplified type systems and generally don't provide for sophisticated scoping of variables and functions. These characteristics make them less suitable for building large, modular applications. Speed is another problem with scripting languages; the high-level, usually source-interpreted nature of these languages often makes them quite slow.

Advocates of individual scripting languages would take issue with some of these generalizations, and no doubt they'd be right in some cases. Scripting languages have improved in recent years—especially JavaScript, which has had an enormous amount of research poured into its performance. But the fundamental tradeoff is undeniable: scripting languages were born as loose, less structured alternatives to systems programming languages and are generally not ideal for large or complex projects for a variety of reasons, at least not today.

Java offers some of the essential advantages of a scripting language: it is highly dynamic, along with the added benefits of a lower-level language. Java has a powerful Regular Expression API that competes with Perl for working with text and language features that streamline coding with collections, variable argument lists, static imports of methods, and other syntactic sugar that make it more concise.

Incremental development with object-oriented components, combined with Java's simplicity, make it possible to develop applications rapidly and change them easily. Studies have found that development in Java is faster than in C or C++, strictly based on language features.[1] Java also comes with a large base of standard core classes for common tasks such as building GUIs and handling network communications. But along with these features, Java has the scalability and software-engineering advantages of more static languages. It provides a safe structure on which to build higher-level frameworks (and even other languages).

As we've already said, Java is similar in design to languages such as Smalltalk and Lisp. However, these languages were used mostly as research vehicles rather than for development of large-scale systems. One reason is that these languages never developed a standard portable binding to operating system services, such as the C standard library or the Java core classes. Smalltalk is compiled to an interpreted bytecode format, and it can be dynamically compiled to native code on the fly, just like Java. But Java improves on the design by using a bytecode verifier to ensure the correctness of compiled Java code. This verifier gives Java a performance advantage over Smalltalk because Java code requires fewer runtime checks. Java's bytecode verifier also helps with security issues, something that Smalltalk doesn't address.

Throughout the rest of this chapter, we'll present a bird's-eye view of the Java language. We'll explain what's new and what's not-so-new about Java and why.

Safety of Design

You have no doubt heard a lot about the fact that Java is designed to be a safe language. But what do we mean by safe? Safe from what or whom? The security features that attract the most attention for Java are those features that make possible new types of dynamically portable software. Java provides several layers of protection from dangerously flawed code as well as more mischievous things such as viruses and Trojan horses. In the next section, we'll take a look at how the Java virtual machine architecture assesses the safety of code before it's run and how the Java *class loader* (the bytecode loading mechanism of the Java interpreter) builds a wall around untrusted classes. These features provide the foundation for high-level security policies that can allow or disallow various kinds of activities on an application-by-application basis.

In this section, though, we'll look at some general features of the Java programming language. Perhaps more important than the specific security features, although often overlooked in the security din, is the safety that Java provides by addressing common design and programming problems. Java is intended to be as safe as possible from the simple mistakes we make ourselves as well as those we inherit from legacy software. The goal with Java has been to keep the language simple, provide tools that have demonstrated their usefulness, and let users build more complicated facilities on top of the language when needed.

Simplify, Simplify, Simplify...

With Java, simplicity rules. Since Java started with a clean slate, it was able to avoid features that proved to be messy or controversial in other languages. For example, Java doesn't allow programmer-defined operator overloading (which in some languages allows programmers to redefine the meaning of basic symbols like + and –). Java doesn't have a source code preprocessor, so it doesn't have things like macros,`#define` statements, or conditional source compilation. These constructs exist in other languages primarily to support platform dependencies, so in that sense, they should not be needed in Java. Conditional compilation is also commonly used for debugging, but Java's sophisticated runtime optimizations and features such as *assertions* solve the problem more elegantly (we'll cover these in Chapter 4).

Java provides a well-defined *package* structure for organizing class files. The package system allows the compiler to handle some of the functionality of the traditional *make* utility (a tool for building executables from source code). The compiler can also work with compiled Java classes directly because all type information is preserved; there is no need for extraneous source "header" files, as in C/C++. All this

means that Java code requires less context to read. Indeed, you may sometimes find it faster to look at the Java source code than to refer to class documentation.

Java also takes a different approach to some structural features that have been troublesome in other languages. For example, Java supports only a single inheritance class hierarchy (each class may have only one "parent" class), but allows multiple inheritance of interfaces. An *interface*, like an abstract class in C++, specifies the behavior of an object without defining its implementation. It is a very powerful mechanism that allows the developer to define a "contract" for object behavior that can be used and referred to independently of any particular object implementation. Interfaces in Java eliminate the need for multiple inheritance of classes and the associated problems.

As you'll see in Chapter 4, Java is a fairly simple and elegant programming language and that is still a large part of its appeal.

Type Safety and Method Binding

One attribute of a language is the kind of *type checking* it uses. Generally, languages are categorized as*static* or *dynamic*, which refers to the amount of information about variables known at compile time versus what is known while the application is running.

In a strictly statically typed language such as C or C++, data types are etched in stone when the source code is compiled. The compiler benefits from this by having enough information to catch many kinds of errors before the code is executed. For example, the compiler would not allow you to store a floating-point value in an integer variable. The code then doesn't require runtime type checking, so it can be compiled to be small and fast. But statically typed languages are inflexible. They don't support collections as naturally as languages with dynamic type checking, and they make it impossible for an application to safely import new data types while it's running.

In contrast, a dynamic language such as Smalltalk or Lisp has a runtime system that manages the types of objects and performs necessary type checking while an application is executing. These kinds of languages allow for more complex behavior and are in many respects more powerful. However, they are also generally slower, less safe, and harder to debug.

The differences in languages have been likened to the differences among kinds of automobiles.[2] Statically typed languages such as C++ are analogous to a sports car:

reasonably safe and fast, but useful only if you're driving on a nicely paved road. Highly dynamic languages such as Smalltalk are more like an off-road vehicle: they afford you more freedom but can be somewhat unwieldy. It can be fun (and sometimes faster) to go roaring through the backwoods, but you might also get stuck in a ditch or mauled by bears.

Another attribute of a language is the way it binds method calls to their definitions. In a static language such as C or C++, the definitions of methods are normally bound at compile time, unless the programmer specifies otherwise. Languages like Smalltalk, on the other hand, are called *late binding* because they locate the definitions of methods dynamically at runtime. Early binding is important for performance reasons; an application can run without the overhead incurred by searching for methods at runtime. But late binding is more flexible. It's also necessary in an object-oriented language where new types can be loaded dynamically and only the runtime system can determine which method to run.

Java provides some of the benefits of both C++ and Smalltalk; it's a statically typed, late-binding language. Every object in Java has a well-defined type that is known at compile time. This means the Java compiler can do the same kind of static type checking and usage analysis as C++. As a result, you can't assign an object to the wrong type of variable or call nonexistent methods on an object. The Java compiler goes even further and prevents you from using uninitialized variables and creating unreachable statements (seeChapter 4).

However, Java is fully runtime-typed as well. The Java runtime system keeps track of all objects and makes it possible to determine their types and relationships during execution. This means you can inspect an object at runtime to determine what it is. Unlike C or C++, casts from one type of object to another are checked by the runtime system, and it's possible to use new kinds of dynamically loaded objects with a degree of type safety. And because Java is a late binding language, it's always possible for a subclass to override methods in its superclass, even a subclass loaded at runtime.

Incremental Development

Java carries all data type and method signature information with it from its source code to its compiled bytecode form. This means that Java classes can be developed incrementally. Your own Java source code can also be compiled safely with classes from other sources your compiler has never seen. In other words, you can write new code that

references binary class files without losing the type safety you gain from having the source code.

Java does not suffer from the “fragile base class” problem. In languages such as C++, the implementation of a base class can be effectively frozen because it has many derived classes; changing the base class may require recompilation of all of the derived classes. This is an especially difficult problem for developers of class libraries. Java avoids this problem by dynamically locating fields within classes. As long as a class maintains a valid form of its original structure, it can evolve without breaking other classes that are derived from it or that make use of it.

Dynamic Memory Management

Some of the most important differences between Java and lower-level languages such as C and C++ involve how Java manages memory. Java eliminates ad hoc “pointers” that can reference arbitrary areas of memory and adds object garbage collection and high-level arrays to the language. These features eliminate many otherwise insurmountable problems with safety, portability, and optimization.

Garbage collection alone has saved countless programmers from the single largest source of programming errors in C or C++: explicit memory allocation and deallocation. In addition to maintaining objects in memory, the Java runtime system keeps track of all references to those objects. When an object is no longer in use, Java automatically removes it from memory. You can, for the most part, simply ignore objects you no longer use, with confidence that the interpreter will clean them up at an appropriate time.

Java uses a sophisticated garbage collector that runs in the background, which means that most garbage collecting takes place during idle times, between I/O pauses, mouse clicks, or keyboard hits. Advanced runtime systems, such as HotSpot, have more advanced garbage collection that can differentiate the usage patterns of objects (such as short-lived versus long-lived) and optimize their collection. The Java runtime can now tune itself automatically for the optimal distribution of memory for different kinds of applications based on their behavior. With this kind of runtime profiling, automatic memory management can be much faster than the most diligently programmer-managed resources, something that some old-school programmers still find hard to believe.

We've said that Java doesn't have pointers. Strictly speaking, this statement is true, but it's also misleading. What Java provides are *references*—a safe kind of pointer. A reference is a strongly typed handle for an object. All objects in Java, with the exception of primitive numeric types, are accessed through references. You can use references to build all the normal kinds of data structures a C programmer would be accustomed to building with pointers, such as linked lists, trees, and so forth. The only difference is that with references, you have to do so in a typesafe way.

Another important difference between a reference and a pointer is that you can't play games (perform pointer arithmetic) with references to change their values; they can point only to specific objects or elements of an array. A reference is an atomic thing; you can't manipulate the value of a reference except by assigning it to an object. References are passed by value, and you can't reference an object through more than a single level of indirection. The protection of references is one of the most fundamental aspects of Java security. It means that Java code has to play by the rules; it can't peek into places it shouldn't and circumvent the rules.

Java references can point only to class types. There are no pointers to methods. People sometimes complain about this missing feature, but you will find that most tasks that call for pointers to methods can be accomplished more cleanly using interfaces and adapter classes instead. We should also mention that Java has a sophisticated Reflection API that actually allows you to reference and invoke individual methods. However, this is not the normal way of doing things. We discuss reflection in Chapter 7.

Finally, we should mention that arrays in Java are true, first-class objects. They can be dynamically allocated and assigned like other objects. Arrays know their own size and type, and although you can't directly define or subclass array classes, they do have a well-defined inheritance relationship based on the relationship of their base types. Having true arrays in the language alleviates much of the need for pointer arithmetic, such as that used in C or C++.

Error Handling

Java's roots are in networked devices and embedded systems. For these applications, it's important to have robust and intelligent error management. Java has a powerful exception handling mechanism, somewhat like that in newer implementations of C++. Exceptions provide a more natural and elegant way to handle errors. Exceptions allow

you to separate error handling code from normal code, which makes for cleaner, more readable applications.

When an exception occurs, it causes the flow of program execution to be transferred to a predesignated “catch” block of code. The exception carries with it an object that contains information about the situation that caused the exception. The Java compiler requires that a method either declare the exceptions it can generate or catch and deal with them itself. This promotes error information to the same level of importance as argument and return types for methods. As a Java programmer, you know precisely what exceptional conditions you must deal with, and you have help from the compiler in writing correct software that doesn’t leave them unhandled.

Threads

Modern applications require a high degree of parallelism. Even a very single-minded application can have a complex user interface—which requires concurrent activities. As machines get faster, users become more sensitive to waiting for unrelated tasks that seize control of their time. Threads provide efficient multiprocessing and distribution of tasks for both client and server applications. Java makes threads easy to use because support for them is built into the language.

Concurrency is nice, but there’s more to programming with threads than just performing multiple tasks simultaneously. In most cases, threads need to be *synchronized* (coordinated), which can be tricky without explicit language support. Java supports synchronization based on the *monitor* and *condition* model—a sort of lock and key system for accessing resources. The keyword `synchronized` designates methods and blocks of code for safe, serialized access within an object. There are also simple, primitive methods for explicit waiting and signaling between threads interested in the same object.

Java also has a high-level concurrency package that provides powerful utilities addressing common patterns in multithreaded programming, such as thread pools, coordination of tasks, and sophisticated locking. With the addition of the concurrency package and related utilities, Java provides some of the most advanced thread-related utilities of any language.

Although some developers may never have to write multithreaded code, learning to program with threads is an important part of mastering programming in Java and something all developers should grasp. SeeChapter 9 for a discussion of this topic.

Scalability

At the lowest level, Java programs consist of *classes*. Classes are intended to be small, modular components. Over classes, Java provides *packages*, a layer of structure that groups classes into functional units. Packages provide a naming convention for organizing classes and a second tier of organizational control over the visibility of variables and methods in Java applications.

Within a package, a class is either publicly visible or protected from outside access. Packages form another type of scope that is closer to the application level. This lends itself to building reusable components that work together in a system. Packages also help in designing a scalable application that can grow without becoming a bird's nest of tightly coupled code.

Safety of Implementation

It's one thing to create a language that prevents you from shooting yourself in the foot; it's quite another to create one that prevents others from shooting you in the foot.

Encapsulation is the concept of hiding data and behavior within a class; it's an important part of object-oriented design. It helps you write clean, modular software. In most languages, however, the visibility of data items is simply part of the relationship between the programmer and the compiler. It's a matter of semantics, not an assertion about the actual security of the data in the context of the running program's environment.

When Bjarne Stroustrup chose the keyword `private` to designate hidden members of classes in C++, he was probably thinking about shielding a developer from the messy details of another developer's code, not the issues of shielding that developer's classes and objects from attack by someone else's viruses and Trojan horses. Arbitrary casting and pointer arithmetic in C or C++ make it trivial to violate access permissions on classes without breaking the rules of the language. Consider the following code:

```
// C++ code
```

```
class Finances {

    private:

        char creditCardNumber[16];

        ...

};


main() {

    Finances finances;


    // Forge a pointer to peek inside the class

    char *cardno = (char *)&finances;

    printf("Card Number = %.16s\n", cardno);

}
```

In this little C++ drama, we have written some code that violates the encapsulation of the `Finances` class and pulls out some secret information. This sort of shenanigan—abusing an untyped pointer—is not possible in Java. If this example seems unrealistic, consider how important it is to protect the foundation (system) classes of the runtime environment from similar kinds of attacks. If untrusted code can corrupt the components that provide access to real resources such as the filesystem, network, or windowing system, it certainly has a chance at stealing your credit card numbers.

If a Java application is to be able to dynamically download code from an untrusted source on the Internet and run it alongside applications that might contain confidential information, protection has to extend very deep. The Java security model wraps three layers of protection around imported classes, as shown inFigure 1-3.

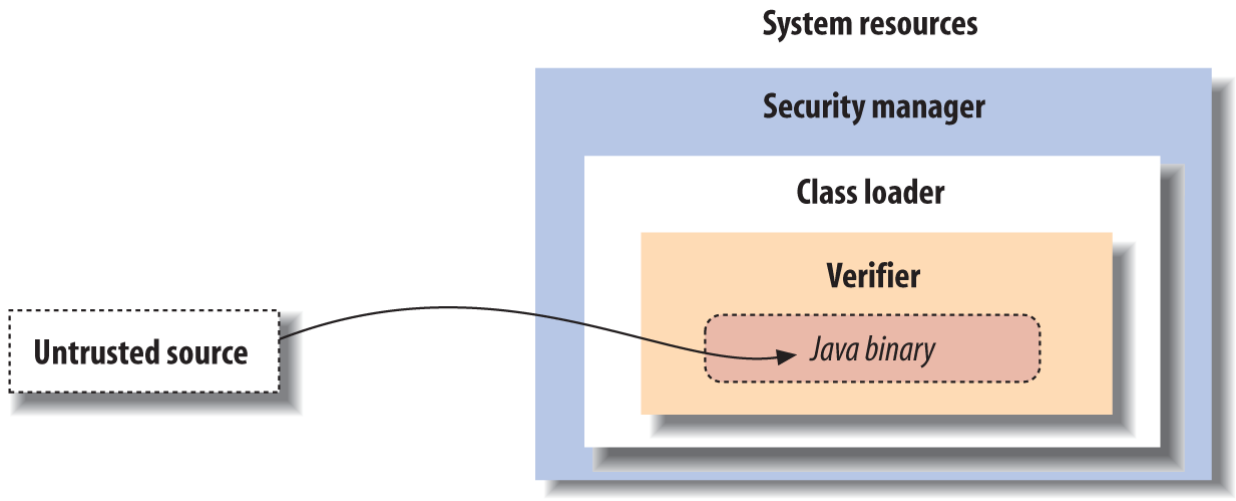

Figure 1-3. The Java security model

At the outside, application-level security decisions are made by a security manager in conjunction with a flexible security policy. A security manager controls access to system resources such as the filesystem, network ports, and windowing environment. A security manager relies on the ability of a class loader to protect basic system classes. A class loader handles loading classes from local storage or the network. At the innermost level, all system security ultimately rests on the Java verifier, which guarantees the integrity of incoming classes.

The Java bytecode verifier is a fixed part of the Java runtime system. Class loaders and security managers (or *security policies* to be more precise), however, are components that may be implemented differently by different applications, such as servers or web browsers. All three of these pieces need to be functioning properly to ensure security in the Java environment.

The Verifier

Java's first line of defense is the *bytecode verifier*. The verifier reads bytecode before it is run and makes sure it is well behaved and obeys the basic rules of the Java language. A trusted Java compiler won't produce code that does otherwise. However, it's possible for a mischievous person to deliberately assemble bad Java bytecode. It's the verifier's job to detect this.

Once code has been verified, it's considered safe from certain inadvertent or malicious errors. For example, verified code can't forge references or violate access permissions on objects (as in our credit card example). It can't perform illegal casts or use objects in

unintended ways. It can't even cause certain types of internal errors, such as overflowing or underflowing the internal stack. These fundamental guarantees underlie all of Java's security.

You might be wondering, isn't this kind of safety implicit in lots of interpreted languages? Well, while it's true that you shouldn't be able to corrupt a BASIC interpreter with a bogus line of BASIC code, remember that the protection in most interpreted languages happens at a higher level. Those languages are likely to have heavyweight interpreters that do a great deal of runtime work, so they are necessarily slower and more cumbersome.

By comparison, Java bytecode is a relatively light, low-level instruction set. The ability to statically verify the Java bytecode before execution lets the Java interpreter run at full speed later with full safety, without expensive runtime checks. This was one of the fundamental innovations in Java.

The verifier is a type of mathematical "theorem prover." It steps through the Java bytecode and applies simple, inductive rules to determine certain aspects of how the bytecode will behave. This kind of analysis is possible because compiled Java bytecode contains a lot more type information than the object code of other languages of this kind. The bytecode also has to obey a few extra rules that simplify its behavior. First, most bytecode instructions operate only on individual data types. For example, with stack operations, there are separate instructions for object references and for each of the numeric types in Java. Similarly, there is a different instruction for moving each type of value into and out of a local variable.

Second, the type of object resulting from any operation is always known in advance. No bytecode operations consume values and produce more than one possible type of value as output. As a result, it's always possible to look at the next instruction and its operands and know the type of value that will result.

Because an operation always produces a known type, it's possible to determine the types of all items on the stack and in local variables at any point in the future by looking at the starting state. The collection of all this type information at any given time is called the *type state* of the stack; this is what Java tries to analyze before it runs an application. Java doesn't know anything about the actual values of stack and variable items at this time; it only knows what kind of items they are. However, this is enough information to enforce the security rules and to ensure that objects are not manipulated illegally.

To make it feasible to analyze the type state of the stack, Java places an additional restriction on how Java bytecode instructions are executed: all paths to the same point in the code must arrive with exactly the same type state.

Class Loaders

Java adds a second layer of security with a *class loader*. A class loader is responsible for bringing the bytecode for Java classes into the interpreter. Every application that loads classes from the network must use a class loader to handle this task.

After a class has been loaded and passed through the verifier, it remains associated with its class loader. As a result, classes are effectively partitioned into separate namespaces based on their origin. When a loaded class references another class name, the location of the new class is provided by the original class loader. This means that classes retrieved from a specific source can be restricted to interact only with other classes retrieved from that same location. For example, a Java-enabled web browser can use a class loader to build a separate space for all the classes loaded from a given URL. Sophisticated security based on cryptographically signed classes can also be implemented using class loaders.

The search for classes always begins with the built-in Java system classes. These classes are loaded from the locations specified by the Java interpreter's *classpath* (see Chapter 3). Classes in the classpath are loaded by the system only once and can't be replaced. This means that it's impossible for an application to replace fundamental system classes with its own versions that change their functionality.

Security Managers

A *security manager* is responsible for making application-level security decisions. A security manager is an object that can be installed by an application to restrict access to system resources. The security manager is consulted every time the application tries to access items such as the filesystem, network ports, external processes, and the windowing environment; the security manager can allow or deny the request.

Security managers are primarily of interest to applications that run untrusted code as part of their normal operation. For example, a Java-enabled web browser can run applets that may be retrieved from untrusted sources on the Net. Such a browser needs to install a security manager as one of its first actions. This security manager then

restricts the kinds of access allowed after that point. This lets the application impose an effective level of trust before running an arbitrary piece of code. And once a security manager is installed, it can't be replaced.

The security manager works in conjunction with an access controller that lets you implement security policies at a high level by editing a declarative security policy file. Access policies can be as simple or complex as a particular application warrants. Sometimes it's sufficient simply to deny access to all resources or to general categories of services, such as the filesystem or network. But it's also possible to make sophisticated decisions based on high-level information. For example, a Java-enabled web browser could use an access policy that lets users specify how much an applet is to be trusted or that allows or denies access to specific resources on a case-by-case basis. Of course, this assumes that the browser can determine which applets it ought to trust. We'll discuss how this problem is addressed through code-signing shortly.

The integrity of a security manager is based on the protection afforded by the lower levels of the Java security model. Without the guarantees provided by the verifier and the class loader, high-level assertions about the safety of system resources are meaningless. The safety provided by the Java bytecode verifier means that the interpreter can't be corrupted or subverted and that Java code has to use components as they are intended. This, in turn, means that a class loader can guarantee that an application is using the core Java system classes and that these classes are the only way to access basic system resources. With these restrictions in place, it's possible to centralize control over those resources at a high level with a security manager and user-defined policy.

Application and User-Level Security

There's a fine line between having enough power to do something useful and having all the power to do anything you want. Java provides the foundation for a secure environment in which untrusted code can be quarantined, managed, and safely executed. However, unless you are content with keeping that code in a little black box and running it just for its own benefit, you will have to grant it access to at least some system resources so that it can be useful. Every kind of access carries with it certain risks and benefits. For example, in the web browser environment, the advantages of granting an untrusted (unknown) applet access to your windowing system are that it can display information and let you interact in a useful way. The associated risks are that the applet may instead display something worthless, annoying, or offensive.

At one extreme, the simple act of running an application gives it a resource—computation time—that it may put to good use or burn frivolously. It's difficult to prevent an untrusted application from wasting your time or even attempting a "denial of service" attack. At the other extreme, a powerful, trusted application may justifiably deserve access to all sorts of system resources (e.g., the filesystem, process creation, network interfaces); a malicious application could wreak havoc with these resources. The message here is that important and sometimes complex security issues have to be addressed.

In some situations, it may be acceptable to simply ask the user to "okay" requests. The Java language provides the tools to implement any security policies you want. However, what these policies will be ultimately depends on having confidence in the identity and integrity of the code in question. This is where digital signatures come into play.

Digital signatures, together with certificates, are techniques for verifying that data truly comes from the source it claims to have come from and hasn't been modified en route. If the Bank of Boofa signs its checkbook application, you can verify that the app actually came from the bank rather than an imposter and hasn't been modified. Therefore, you can tell your browser to trust applets that have the Bank of Boofa's signature.

A Java Road Map

With everything that's going on, it's hard to keep track of what's available now, what's promised, and what has been around for some time. The following sections constitute a road map that imposes some order on Java's past, present, and future.

The Past: Java 1.0–Java 1.6

Java 1.0 provided the basic framework for Java development: the language itself plus packages that let you write applets and simple applications. Although 1.0 is officially obsolete, there are still a lot of applets in existence that conform to its API.

Java 1.1 superseded 1.0, incorporating major improvements in the Abstract Window Toolkit (AWT) package (Java's original GUI facility), a new event pattern, new language facilities such as reflection and inner classes, and many other critical features. Java 1.1 is the version that was supported natively by most versions of Netscape and Microsoft Internet Explorer for many years. For various political reasons, the browser world was frozen in this condition for a long time. This version of Java is still considered a sort of

baseline for applets, although even this will fall away as Microsoft drops support for Java in its platforms.

Java 1.2, dubbed “Java 2” by Sun, was a major release in December 1998. It provided many improvements and additions, mainly in terms of the set of APIs that were bundled into the standard distributions. The most notable additions were the inclusion of the Swing GUI package as a core API and a new, full-fledged 2D drawing API. Swing is Java’s advanced user interface toolkit with capabilities far exceeding the old AWT’s. (Swing, AWT, and some other packages have been variously called the JFC, or Java Foundation Classes.) Java 1.2 also added a proper Collections API to Java.

Java 1.3, released in early 2000, added minor features but was primarily focused on performance. With version 1.3, Java got significantly faster on many platforms and Swing received many bug fixes. In this timeframe, Java enterprise APIs such as Servlets and Enterprise JavaBeans also matured.

Java 1.4, released in 2002, integrated a major new set of APIs and many long-awaited features. This included language assertions, regular expressions, preferences and logging APIs, a new I/O system for high-volume applications, standard support for XML, fundamental improvements in AWT and Swing, and a greatly matured Java Servlets API for web applications.

Java 5, released in 2004, was a major release that introduced many long-awaited language syntax enhancements including generics, typesafe enumerations, the enhanced for-loop, variable argument lists, static imports, autoboxing and unboxing of primitives, as well as advanced metadata on classes. A new concurrency API provided powerful threading capabilities, and APIs for formatted printing and parsing similar to those in C were added. RMI has also been overhauled to eliminate the need for compiled stubs and skeletons. There were also major additions in the standard XML APIs.

Java 6, released in late 2006, was a relatively minor release that added no new syntactic features to the Java language, but bundled new extension APIs such as those for XML and web services.

The Present: Java 7

This book includes all the latest and greatest improvements through the final release of Java 7. This release adds some minor language syntax enhancements such as those to

improve exception handling and resource management. It also includes some major API updates, such as a completely new filesystem API and additions to many others.

This edition of the book is the first since the Java 5 release and therefore has been completely overhauled to incorporate all of the changes from the Java 6 and Java 7 releases.

Here's a brief overview of the most important features of the current core Java API:

JDBC (Java Database Connectivity)

: A general facility for interacting with databases (introduced in Java 1.1).

RMI (Remote Method Invocation)

: Java's distributed objects system. RMI lets you call methods on objects hosted by a server running somewhere else on the network (introduced in Java 1.1).

Java Security

: A facility for controlling access to system resources, combined with a uniform interface to cryptography. Java Security is the basis for signed classes, which were discussed earlier.

JFC (Java Foundation Classes)

: A catch-all for a number of features, including the Swing user interface components; "pluggable look and feel," which means the ability of the user interface to adapt itself to the look and feel of the platform you're using; drag and drop; and accessibility, which means the ability to integrate with special software and hardware for people with disabilities.

Java 2D

: Part of JFC; enables high-quality graphics, font manipulation, and printing.

Internationalization

: The ability to write programs that adapt themselves to the language the user wants to use; the program automatically displays text in the appropriate language (introduced in Java 1.1).

JNDI (Java Naming and Directory Interface)

> A general service for looking up resources. JNDI unifies access to directory services, such as LDAP, Novell's NDS, and others.

The following are "standard extension" APIs. Some, such as those for working with XML and web services, are bundled with the standard edition of Java; some must be downloaded separately and deployed with your application or server.

JavaMail

> A uniform API for writing email software.

Java 3D

> A facility for developing applications with 3D graphics.

Java Media

> Another catch-all that includes Java 2D, Java 3D, the Java Media Framework (a framework for coordinating the display of many different kinds of media), Java Speech (for speech recognition and synthesis), Java Sound (high-quality audio), Java TV (for interactive television and similar applications), and others.

Java Servlets

> A facility that lets you write server-side web applications in Java.

Java Cryptography

> Actual implementations of cryptographic algorithms. (This package was separated from Java Security for legal reasons.)

JavaHelp

> A facility for writing help systems and incorporating them in Java programs.

Enterprise JavaBeans

> A component architecture for building distributed server-side applications.

Jini

An interesting distributed component technology that is designed to enable distributed computing, discovery, and rendezvous of devices ranging from software tools to hardware and household appliances.

XML/XSL

Tools for creating and manipulating XML documents, validating them, mapping them to and from Java objects, and transforming them with stylesheets.

Web services

Tools for creating and deploying Java-based SOAP web services.

In this book, we'll try to give you a taste of as many features as possible; unfortunately for us (but fortunately for Java software developers), the Java environment has become so rich that it's impossible to cover everything in a single book.

The Future

Changes in Java have become less frequent as Java has matured over the years, but Java continues to be one of the most popular platforms for application development. This is especially true in the areas of web services, web application frameworks, and XML tools. While Java has not dominated mobile platforms in the way it seemed destined to, the Java language and core APIs are used to program for Google's Android mobile OS, which is used on billions of devices around the world. In the Microsoft camp, the Java-derived C# language has taken over much .NET development and brought the core Java syntax and patterns to those platforms.

Probably the most exciting areas of change in Java today are found in the trend toward lighter weight, simpler frameworks for business and the integration of the Java platform with dynamic languages for scripting web pages and extensions. There is much more interesting work to come.

Availability

You have several choices for Java development environments and runtime systems. Oracle's Java Development Kit (JDK) is available for Mac OS X, Windows, and Linux. Visit Oracle's Java website at for more information about obtaining the latest JDK. This book's online content is available athttp://oreil.ly/Java_4E.

There is also a whole array of popular Java Integrated Development Environments. We'll discuss two in this book: IBM's Eclipse and the Oracle NetBeans IDE. These all-in-one development environments let you write, test, and package software with advanced tools at your fingertips. While Eclipse is unquestionably the most popular and is open source, this author's preferred IDE is Intellij IDEA by JetBrains, which now also has a free community edition.

[1] See, for example, G. Phipps, "Comparing Observed Bug and Productivity Rates for Java and C++,"*Software—Practice & Experience*, volume 29, 1999.

[2] The credit for the car analogy goes to Marshall P. Cline, author of the C++ FAQ.